

User-Defined Datatypes

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Even more powerful than the idea of recursion is the idea of abstraction—which is a lot easier to implement if you can make your own types. Your goal after this lecture is to...

- Be able to describe the record type and the disjoint type.
- Be able to give an example of how each one is used.
- Be able to draw a memory diagram that is the result of allocating a variable of a disjoint type.
- Be able to use the `match/with` syntax to deconstruct a disjoint type.
- Be able to describe how data-structures in a functional language “recycle” data from previous versions of the structure when a modification has been made.

- Complex numbers have the form $a + bi$, where $i \equiv \sqrt{-1}$
- Addition: $(a + bi) + (c + di) = (a + c) + (b + d)i$
- Multiplication: $(a + bi) \times (c + di) = ac - bd + (ad + bc)i$

```
1 # let cadd (a,b) (c,d) = a +. c, b +. d
2 # let cmul (a,b) (c,d) = a *. c -. b *. d,
3                        a *. d +. b *. c
4 # cmul (3.0,4.0) (1.0,4.4);;
5 - : float * float = -14.6, 17.2
```

We could use tuples to represent complex numbers, like this. (What are the types of these functions?) Why might this be a bad idea?

`type name = { name : type [; name : type ...] }`

Diagram illustrating the relationship between the `name` field in the record type definition and the `Name` and `Labels` terms. Dotted arrows point from `Name` and `Labels` to the `name` field in the record definition.

```
1 # type complex = { re : float; im : float };;  
2 type complex = { re : float; im : float; }  
3 # let cadd x y = { re = x.re +. y.re;  
4                  im = x.im +. y.im };;  
5 val cadd : complex -> complex -> complex = <fun>
```

- Fields are accessed with dot notation.
- Note that *field names must be unique*.

1. Write a function to multiply complex numbers. Use the type definition we had before.
2. What would the type definition for a phone book entry look like? Assume we want to store a name, address, and phone number, all as strings.

1. Write a function to multiply complex numbers. Use the type definition we had before.

```
1 let mult a b = { re = a.re * b.re - a.im * b.im;  
2                  im = a.re * b.im + b.re * a.im;
```

2. What would the type definition for a phone book entry look like? Assume we want to store a name, address, and phone number, all as strings.

```
1 type address = {name : string; address : string;  
2                phone : string };;
```

`type name = Name [of type] [| Name [of type] ...]`

Diagram illustrating the components of a type definition:

- Name** points to the *Name* in *Name* [*of type*].
- Constructors** points to the `[` in *Name* [*of type*].
- Arguments** points to the `[` in *Name* [*of type*] ... and the `[` in *Name* [*of type*].

- Note: Constructor names must be capitalized.
- Constructor names also must be unique.

```
1 # type contest = Rock | Scissors | Paper
2 # type velocity = MeterPerSecond of float
3                   | FeetPerSecond of float;;
```

```
1 # let winner a b =  
2   match a,b with  
3     | Rock,Scissors | Scissors,Paper | Paper,Rock  
4       -> "Player 1"  
5     | Rock,Paper | Scissors,Rock | Paper,Scissors  
6       -> "Player 2"  
7     | _ -> "Tie";;  
8 val winner : contest -> contest -> string = <fun>  
9 # let thrust vel = If only NASA had used OCaml!  
10   match vel with  
11     | FeetPerSecond x    -> x /. 3.28  
12     | MetersPerSecond x -> x;;  
13 val thrust : velocity -> float = <fun>
```

```
1 # type foo = Foo of int;;
2 # let showfoo (Foo n) = n;;
```

```
1 # #use "foo.ml";; This loads the file foo.ml
2 # let t1 = Foo 5;;
3 # showfoo t1;;
4 - : int = 5
5 # #use "foo.ml";; At this point you reload....
6 # showfoo t1;;
7 Characters 8-10:
8 This expression has type foo but is here used
9 with type foo
```

Why did this happen?

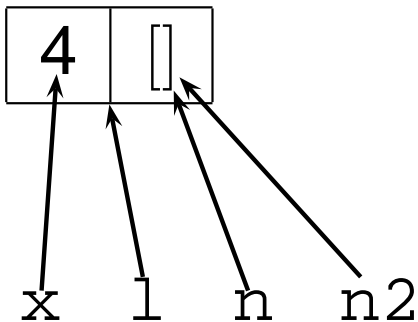
```
1 # type mylist = Cons of int * mylist
2           | Nil
3 # let rec mklist lst =
4     match lst with
5     | [] -> Nil
6     | x::xs -> Cons (x, mklist xs);;
7 # let l1 = mklist [2;3;4];;
8 val l1 : mylist = Cons (2, Cons (3, Cons (4, Nil)))
```

- A recursive type without a recursive case is not really recursive.
- A recursive type without a base case is useless. Unless you're using Haskell, which supports infinite data-structures. But we're not going to talk about that here.

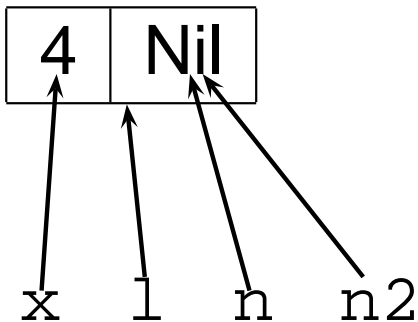
- When a type constructor is invoked, it causes memory to be allocated.
 - Writing an integer
 - Writing `[]` or `::`
 - Using `::` or `Cons`
- Writing down a variable does not cause memory to be allocated.

```
1 # let x = 4;;   allocates 4
2 # let n = [];;  allocates empty list
3 # let n2 = n;;  does NOT allocate memory
4 # let l = x::n;; A cons cell is allocated,
5               but not the 4 or the empty list
```

```
1 # let x = 4;;    allocates 4
2 # let n = [];;   allocates empty list
3 # let n2 = n;;   does NOT allocate memory
4 # let l = x::n;; A cons cell is allocated,
5   but not the 4 or the empty list
```



```
1 # let x = 4;;   allocates 4
2 # let n = Nil;; allocates Nil
3 # let n2 = n;;  does NOT allocate memory
4 # let l = Cons (x,n);; A cons cell is allocated,
5   but not the 4 or a Nil
```



How would you write `length` and `inclist` for this type?

```
1 # let rec length lst =
2     match lst with
3     | Nil -> 0
4     | Cons(x,xs) -> 1 + length xs;;
5 # length l1
6 - : int = 3
7 # let rec inclist lst =
8     match lst with
9     | Nil -> Nil
10    | Cons(x,xs) -> Cons(x+1,inclist xs);;
11 # inclist l1;;
12 - : mylist = Cons (3, Cons (4, Cons (5, Nil)))
```

```
1 # let rec reverse lst =  
2   let rec aux lst acc =  
3     match lst with  
4     | Nil -> acc  
5     | Cons(x,xs) -> aux xs (Cons(x,acc)) in  
6   aux lst Nil;;  
7 # reverse l1;;  
8 - : mylist = Cons (4, Cons (3, Cons (2, Nil)))
```

OCaml supports parametric polymorphism, like templates in C++ or generics in Java.

```
1 # type 'a mylist = Cons of 'a * 'a mylist
2           | Nil
3 # let rec length xx =
4     match xx with
5     | Nil -> 0
6     | Cons (_,xs) -> 1 + length xs;;
7 val length : 'a mylist -> int = <fun>
8 # length (Cons(3,Cons(4,Nil)));;
9 - : int = 2
10 # length (Cons("Hi",Cons("There",Nil)));;
11 - : int = 2
```


1. Write the type for a binary search tree. In this version, let a “node” be something that has data, and a “leaf” does not have any data. Use parametric polymorphism.
2. Write `find` and `add`.

Let's take these one at a time.

Write the type for a binary search tree.

```
1 type 'a bst = Node of 'a * 'a BST * 'a BST
2           | Leaf
```

Now write `find`.

```
1 let rec find a xx =  
2   match xx with  
3   | Node (x,l,r) when x=a -> true  
4   | Node (x,l,r) -> if x<a then find a l  
5                       else find a r  
6   | Leaf -> false
```

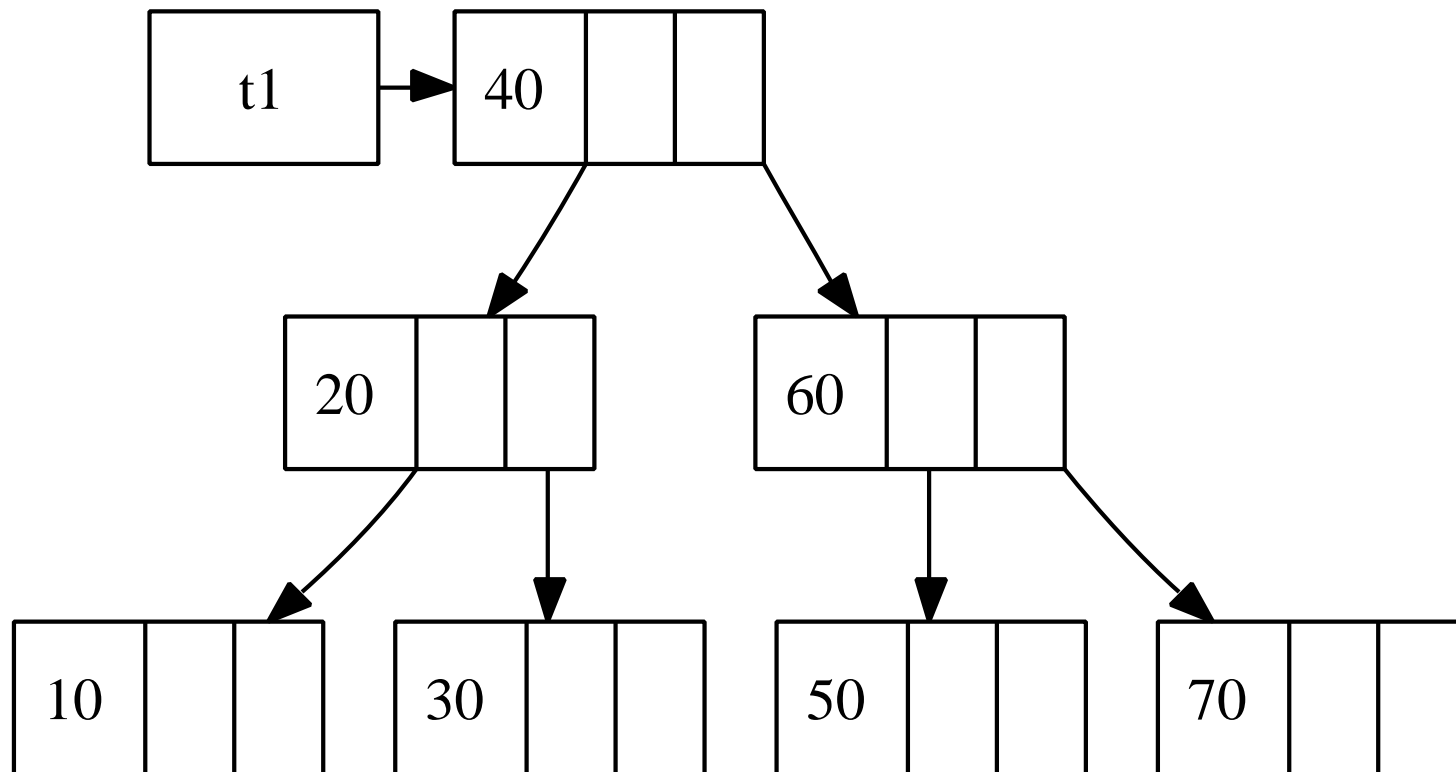
Now write `add`. If the element is already added, return the value unchanged.

```
1 let rec add a xx =  
2   match xx with  
3   | Node (x,l,r) when x=a -> xx  
4   | Node (x,l,r) -> if x<a then Node(x, add a l, r)  
5                       else Node(x, l, add a r)  
6   | Leaf -> Node(a,Leaf,Leaf)
```

What will be the result of

```
1 let t1 = add 10 (add 30 (add 50 (add 70  
2   (add 20 (add 60 (Node 40 Leaf Leaf)))))) ; ;
```

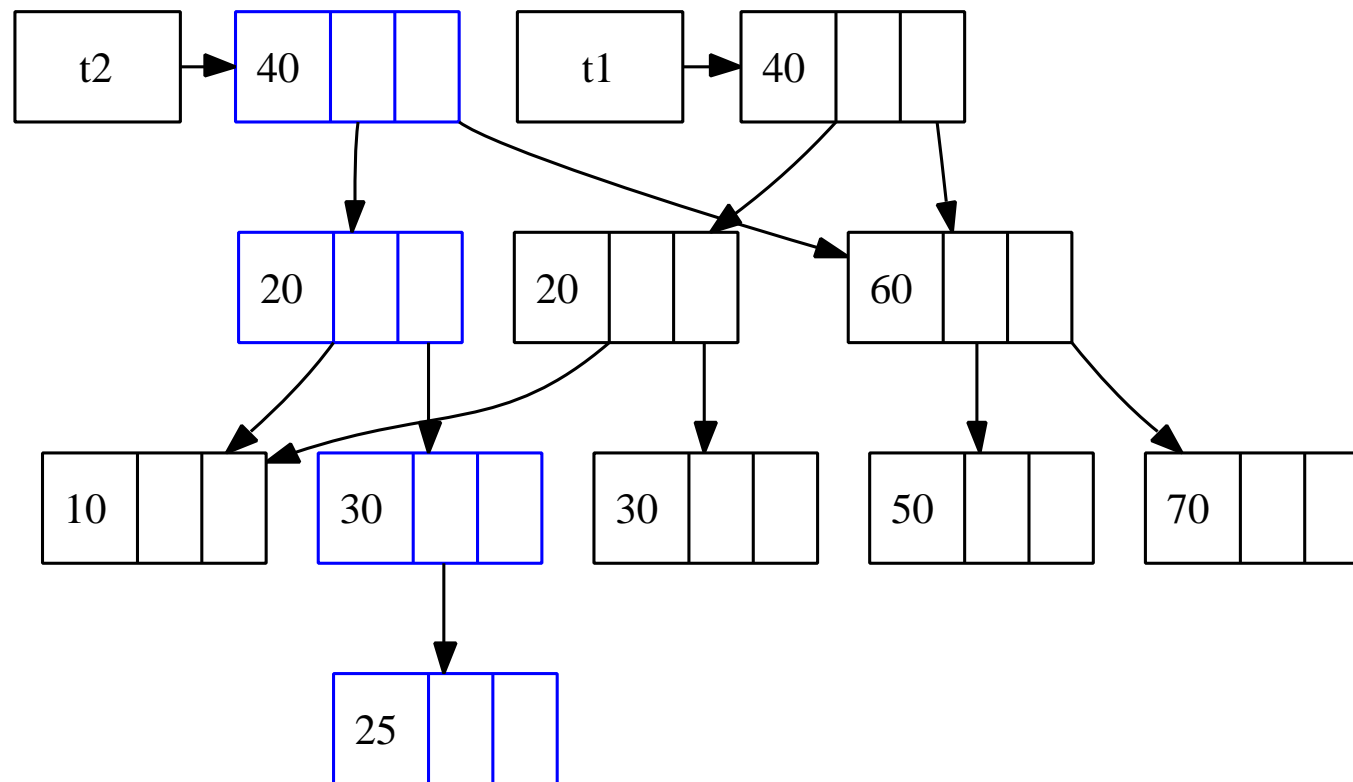
Before:



Now execute:

```
1 let t2 = add 25 t1;;
```

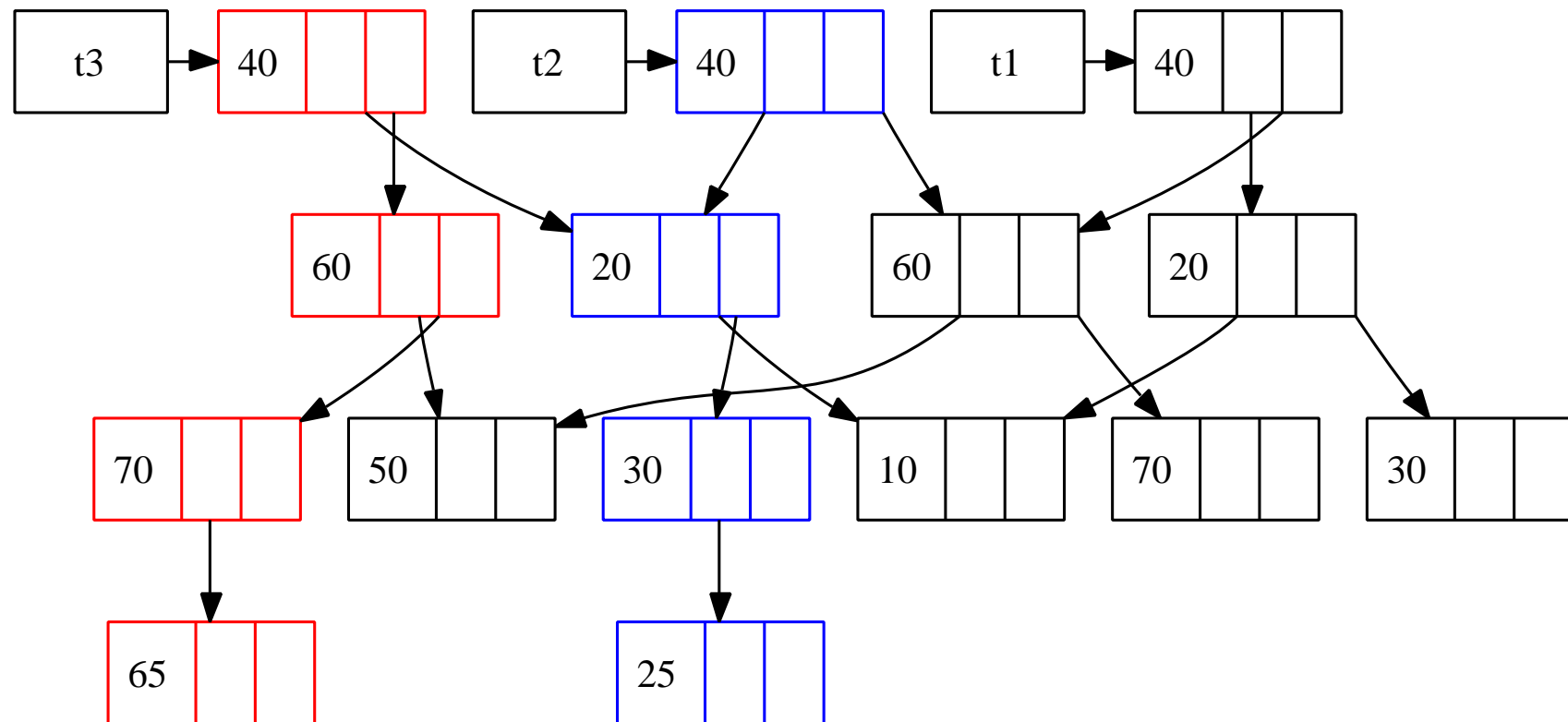
Before:



Now execute:

```
1 let t3 = add 65 t2;;
```

Before:



- This technique is called *functional-style updating*.
- Advantage: multiple versions of the data-structure can exist. This allows multiple versions, and consistent multiprocessing. Used to implement multi-level undo.
- Disadvantage: it takes more memory.


```
1 # type 'a option = Some of 'a | None;;
2 # let rec getItem key lst =
3     match lst with
4     | [] -> None      note the type variables!
5     | (k,v)::xs -> if key = k then Some v
6                     else getItem key xs;;
7 # getItem 3 [2,"french hens"; 3,"turtle doves"];;
8 - : string option = Some "turtle doves"
9 # getItem 5 [2,"french hens"; 3,"turtle doves"];;
10 - : string option = None
```

What options would you have for this function if you did *not* have an option type?